852L1-DataCoding-Lecture6

November 13, 2020

1 Lecture 6: Introduction to Python and Data Types

1.1 Welcome to Python

From this lecture onwards, we will learn the Python programming language and the related packages for Data Science and Visualisation.

1.1.1 Why Python?

So you've decided that you want to learn how to code. Among the many existing programming languages, why should you choose Python?

There are many reasons:

- It's easy to learn and use (compared to C, C++, Fortran, etc.)
- General purpose language:
  - Data science, data visualiation, financial and economic modelling, websites creation, etc.
- It is designed with readability in mind
- It is the most popular programming language in use
  - Check here
- High demand for Python skills in the industry
  - Check the list of jobs vacancies on Linkedin

1.2 Outline of Python lectures

Over the next weeks, we will cover the following topics

- Introduction to Python: basic commands and syntax
- Data types
  - Numbers
  - Strings
  - Lists, tuples, and dictionaries
- Control flow
  - Conditional statements (if…elseif)
  - Loops
- Functions
- Importing and using packages
- Python for Data Science and Visualisation
  - Pandas
  - Matplotlib

1.3 Your first Python program!

In every programming language, the novice programmer always starts with the famous Hello World program. This simply consists of instructing the computer to display to the screen the following sentence:

> Hello, World!

Python is no different … so let's do that!

```
[1]: # Tell Python to print Hello, World! to the screen
     print("Hello, World!")
```

```
Hello, World!
```

That's it! Our first Python program

That was easy, wasn't it?

Let's have a look at our program once again. We can see **two** distinct pieces: 1. A **comment** 2. A **command**

In the first line, we have a **comment**:

```
# Tell Python to print Hello, World! to the screen
```

Comments are entire lines or parts of text within our code that are **not interpreted** by the computer. Whenever Python "reads" our code, it **completely ignores** this part of our code entirely.

Let's revisit out first program above:

```
[2]: # Tell Python to print Hello, World! to the screen
     # print("Hello, World!")
```

When I run the above cell, Python does not print anything to screen. **This is the expected behaviour**: *the above program only contains comments, and not actual codes.*

1.3.1 Why comments?

Comments are useful to document your code. They explain what is going on to another person that is reading your program, or your future-self that reads your own code in three months!

Think of comments as little post-in notes that you put there to remind you what each line is doing.

Comments start with the hash character **#**. They will be highlighted by a different color:

```
# This is a comment!
```

Moreover, you can occasionally use comments to select which part of code to run:

```
[3]: # Tell Python to print Hello, World! to the screen
     print("Hello, World!")
     print("Hello, Luca!")
```

```
Hello, World!
Hello, Luca!
```

You can use comments on the same line as actual code. For instance:

```
[4]: # Below is an example of mixing code and comments on the same line
     print("Hello, planet Earth!")        # This line prints "Hello, planet Earth!"␣
      ↪to screen
```

```
Hello, planet Earth!
```

1.4 Can we write commands in plain English? Welcome Syntax!

Have a look at our command once again:

`print("Hello, World!")`

Notice that it is written in a **very specific way**:

- `print()` has to be followed by opening `(` and closing `)` parentheses
- `"Hello, World!"` is contained within quotes `" "`

When writing Python code, we need to follow its **syntax**:

- This is the set of rules of writing that allows Python to understand and interpret our code

Any deviation from the specified syntax confuses Python, which starts complaining.

Let's see what happens if we "violate the rules":

```
[5]: # Example 1 of incorrect syntax
     print("Hello, World!"
```

```
      File "<ipython-input-5-0bb8810d7184>", line 2
    print("Hello, World!"
                         ^
SyntaxError: unexpected EOF while parsing
```

```
[6]: # Example 2 of incorrect syntax
     print("Hello, World!)
```

```
      File "<ipython-input-6-149a2a6a74d6>", line 2
    print("Hello, World!)
                         ^
SyntaxError: EOL while scanning string literal
```

Whenever we violate the Python syntax, the program does not run correctly and reports an error.

This is called a **syntax error**: - Syntax errors comes with a brief explanation. You should read this carefully in order to fix your code.

1.5 Python as a calculator

Python is an incredibly versatile and powerful programming language. It will allow us to perform complex and computer-intesive task such as reading, summarising, and visualising datasets, as well as writing programs that performs specific tasks.

Before we jump to that, we need to start with the basics. The easiest thing that we can do with Python is to use it a **calculator**.

Similarly to what you would do with a standard pocket calculator, Python can work with **numbers** and perform the following operations:

Operation	Operator
Addition	`+`
Subtraction	`-`
Multiplication	`*`
Division	`/`
Floor division	`//`
Modulo/remainder	`%`
Exponentiation	`**`

Let's have a look at each of those with some examples:

[7]:
```python
# Sum the numbers 6 and 4
6 + 4
```

[7]:
```
10
```

[8]:
```python
# Subtract 3 from 10
10 - 3
```

[8]:
```
7
```

[9]:
```python
# Subtraction works with negative results too. Subtract 10 from 3
3 - 10
```

[9]:
```
-7
```

[10]:
```python
# Multiply 5 by 2
5 * 2
```

[10]:
```
10
```

As we have seen, all of the above operations are the familiar ones that we use everyday. Morover, these operations return whole numbers, i.e. numbers without decimal points.

These numbers are called **integers**:

> **Integer** is a built-in **data type** in Python

To confirm this we can use the `type()` command:

[11]:
```
# Check that 2 is an integer using type()
type(2)
```

[11]:
```
int
```

Integers is the first of many data types that we will use. Another one is a **float**

> A **float** is a built-in data type used to represent fractions and numbers with decimal points

Floats are usually the result of **division** operations.

Let's see an example:

[12]:
```
# Divide 20 by 3
20 / 3
```

[12]:
```
6.666666666666667
```

As you can see, the result contains decimals points. Indeed, this is a **float**:

[13]:
```
# Check that 20 / 3 is a float using type()
type(20 / 3)
```

[13]:
```
float
```

There are two other types of "divisions": - One that discards the decimal points altogheter (floor division) - One that returns the remainder of the division only (modulo)

[14]:
```
# Keep only the integer part of the division
20 // 3
```

[14]:
```
6
```

[15]:
```
# Keep only the remainder of the division
20 % 3
```

[15]:
```
2
```

1.5.1 Operators precendence

Different operators can be combined together, that is you can sum, subtract, multiply, and divide numbers in a single operation.

When doing so we need to keep in mind the precedence of operators. Operators have a precedence order similar to standard arithmetic and mathematical operations that you do with pen and paper.

```
[16]: # Division, then multiplication, and finally sum
      1 + 6 / 2 * 3       # --> should give 10
```

```
[16]: 10.0
```

What if I want to do the operations in a different order?

This could be easily achieved by using parentheses `(` and `)`

```
[17]: # Use parentheses to sum 1 and 6 first
      # then multiply 2 by 3
      # and finally divide the results
      (1 + 6) / (2 * 3)       # --> should be 1.66666667
```

```
[17]: 1.1666666666666667
```

1.6 Storing results: Variables

Great, we can do basic maths with Python!

What if we want to do more complex operations, for example by **storing** our results and use it later on in our code?

We can do that by using **variables**

> **Variables** allows us to give a "name" to an **instance** of data type (here numbers) or the result of an operation.
>
> Variables names are remembered by Python and can be **called** later on in the program

1.6.1 Example: cost of a basket

Let's say that we want to compute the cost of the following basket, given the prices and quantity purchased:

Good	Price	Quantity
Cola	1.25	5
Water	0.75	7
Chocolate	3	4

Of course, we need to first compute the cost of individual goods and later sum them together

```
[18]: # Compute cost of individual goods and store them in a variables
      cost_cola      = 1.25 * 5
      cost_water     = 0.75 * 7
      cost_chocolate = 3 * 4
```

```
# You can print the cost of the goods to check if they're correct
print(cost_cola)
print(cost_water)
print(cost_chocolate)
```

```
6.25
5.25
12
```

```
[19]: # Finally, use the previously created variables to store the cost of the basket
cost_basket = cost_cola + cost_water + cost_chocolate
```

```
[20]: # Notice that when you create a new variable, it does not show automatically
# Use the print() function to do this
print(cost_basket)
```

```
23.5
```

1.6.2 Warnings!!!

Variables are **case sensitive**, that is lowercase and uppercase characters matter.

> For instance, the variable `Cost_basket` is different from `cost_basket`.

If we try to use the capitalised variables, it would give us an error. Indeed, the variable does **not** exist!

```
[21]: # Try to print a variable that does not exists
print(Cost_basket)
```

```
           ␣
 ↪---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call␣
 ↪last)

        <ipython-input-21-bfc9586a9538> in <module>
          1 # Try to print a variable that does not exists
    ----> 2 print(Cost_basket)


        NameError: name 'Cost_basket' is not defined
```

# Scripts

Notice that we are doing operations sequentially:

1. Create and store variables with individual goods cost
2. Use these variables to create a new variabile with total cost

3. Print the result out

Python, as other programming languages, allows you write all of these operations and commands in text file, which can be read and run by the interpreter. These text files are called **scripts**.

> Python scripts are text files with extension .py
>
> Python scripts are read by the Python interpreter and invoke all operations sequentially when these are runned.

For example, the file `cost_of_basket.py` stores all operations done above in a script. You can invoke and run this script in this Jupyter notebook

[22]:
```
# Invoke and run the Python script that compute the cost of a basket
%run cost_of_basket.py
```

```
23.5
```

1.7 Working with text: Strings

Let's have a final look at our very first program:

```
# Tell Python to print Hello, World! to the screen
print("Hello, World!")
```

As we previously mentioned briefly, `"Hello, World!"` is a **string**. This is the second Python built-in data type that we introduce.

> A **string** is a Python data type that stores a *sequence of characters*

Textual data, and thus strings, can be our main focus of analysis or can be used to document the output of our program along with numbers.

### Creating strings There are many way to create strings. The two most used are:

- Using single quotes `''`
- Using double quotes `""`

As with numbers, strings can be stored in variables and later printed. Let's see them in action!

[23]:
```
# Create, store, and print a string using single quotes
my_first_string = 'This is a string'
print(my_first_string)
```

```
This is a string
```

[24]:
```
# Create, store, and print a string using double quotes
my_second_string = "This is another string"
print(my_second_string)
```

```
This is another string
```

[25]:
```
# Very long string example. Use parentheses to encapsulate the string and store␣
↪it in variables
```

```
long_sentence = ("Sometimes you have a very long sentence "
                 "that does not fit on a single line and "
                 "you might want to define it over several lines. "
                 "Nevertheless this is stored, and printed, as a "
                 "SINGLE line")
print(long_sentence)
```

```
Sometimes you have a very long sentence that does not fit on a single line and
you might want to define it over several lines. Nevertheless this is stored, and
printed, as a SINGLE line
```

Some other times you want to display a sentence or a sequence of sentences over **multiples** lines. There are many ways to do that. Here's two:

- Define your strings using triple quotes `""""""`
- Use the **escape character** `\n`

[26]:
```
# Define a string with triple quotes in order to be displayed over several lines
multiple_lines_string = """This
sentence
spans
multiple
lines"""

print(multiple_lines_string)
```

```
This
sentence
spans
multiple
lines
```

[27]:
```
# Use the escape character \n to break a sentence over multiple lines
long_sentence_again = ("Sometimes you have a very long sentence\n"
                 "that you want to display over several lines.\n"
                 "To do that use the escape character '\\n' where you␣
 ↪want\nyour line to be split.")
print(long_sentence_again)
```

```
Sometimes you have a very long sentence
that you want to display over several lines.
To do that use the escape character '\n' where you want
your line to be split.
```

### Combining strings We have seen how to create strings and store them in variables. As with numbers, you can use these variables to create new strings.

For instance, you can combine two or more strings to create a longer string. This operation is called **string concatenation**.

We can do that with two operators:

- `+` joins two or more strings together
- `*` repeats the same string multiple times

Clearly, this is your favourite module of this year. You want Python to know that.

[28]:
```
# Define three strings and store them in separate variables
module_code = "852L1"
module_name = "Data Processing, Coding, & Visualisation"
year        = "2020"

# Create a new variable that uses the strings above
fav_module = module_code + " " + module_name + " is my favourite module of " +␣
 ↪year

# Print it
print(fav_module)
```

```
852L1 Data Processing, Coding, & Visualisation is my favourite module of 2020
```

[29]:
```
# Show your lecturer how much you're excited about it
oh_yeah = 10 * "O" + "H YEAH! " + 15 * " "
print(oh_yeah)
```

```
OOOOOOOOOOH YEAH!
```

1.7.1 String formatting

Combining strings with `+` and `*` works, but there is a better way to do that. That is **string formatting** and more specifically **f-strings**

> A **f-string** is a string that is *prepended* by the character `f`

Example:

`f"This is an f-string"`

What's so special about f-string?

f-string can be used along with a *placeholder* (identified by curly braces `{}`) which can contain **variables** and/or **expressions**.

Note: f-strings are available with versions of Python 3.6 or never

[30]:
```
# Repeat that this is your favourite module with f-strings
fav_module = f"{module_code} {module_name} is my favourite module of {year}"
print(fav_module)
```

```
852L1 Data Processing, Coding, & Visualisation is my favourite module of 2020
```

### Combining strings and numbers

Formatted strings allow us to combine strings with other data types, such as numbers.

Let's consider again our program that computes the cost of a basket. If we print out the variable `cost_basket`, we'll only see a number that represent the total cost of the basket.

That's not very informative …

[31]:
```
cost_basket
```

[31]:
```
23.5
```

It would be much better if we can see the entire receipt from our grocery shopping

[32]:
```
# Create a variable the contains all information about the basket
receipt = (f"Cola:       £ {cost_cola}\n"
           f"Water:      £ {cost_water}\n"
           f"Chocolate:  £ {cost_chocolate}\n"
            "              ---- \n"
           f"Total cost: £ {cost_basket}"
          )

# Print receipt
print(receipt)
```

```
Cola:       £ 6.25
Water:      £ 5.25
Chocolate:  £ 12
              ----
Total cost: £ 23.5
```

1.8 Extracting text: indexing and slicing

Sometimes we want to extract text from existing variables. That is, given some text stored in a string variable we want to

- Access a specific character of that text
- Extract a subset of the text, i.e. a sub-string

We can do that with two operations: **indexing** and **slicing**.

1.8.1 String indexing

Indexing allows us to access a specific character of a given string. This is done by identifying the position of that character in the string.

Given any string, e.g. `"Coding"`, each character is associated to an **index** that specifies that position of each character:

	C	o	d	i	n	g	
→	0	1	2	3	4	5	
	-6	-5	-4	-3	-2	-1	←

You can access that character by doing

`string_name[index]`

A few examples below:

```
[33]: c = "Coding"
```

Going forward:

```
[34]: # First element
      c[0]
```

```
[34]: 'C'
```

```
[35]: # Third element
      c[2]
```

```
[35]: 'd'
```

```
[36]: # Last element
      c[5]
```

```
[36]: 'g'
```

Going backwards:

```
[37]: # Last element
      c[-1]
```

```
[37]: 'g'
```

```
[38]: # Third element
      c[-4]
```

```
[38]: 'd'
```

```
[39]: # First element
      c[-6]
```

```
[39]: 'C'
```

Notice that it is not allowed to use an index beyond the length of the string:

```
[40]: # Index beyond length of string
      c[6]
```

```
         ␣
↪---------------------------------------------------------------------------
```

```
        IndexError                                Traceback (most recent call␣
 ↪last)

        <ipython-input-40-0af7d9d61526> in <module>
          1 # Index beyond length of string
    ----> 2 c[6]


        IndexError: string index out of range
```

### String slicing In addition to extracting specific characters from a string, Python allows us to extract *substrings*. This is done with **slicing**.

If we have a string named `string_name`, then the following expression

`string_name[start:end]`

returns a sub-string of `string_name` with characters from `start` (included) to `end` (**not** included)

Warning: The `end` character is **not** included.

While this might sound unintuitive, the output of the slicing operation makes sense. Indeed, we obtain a substring of length equal to `end - start`

Let's reuse our `"Coding"` string for examples:

	C	o	d	i	n	g	
→	0	1	2	3	4	5	
	-6	-5	-4	-3	-2	-1	←

[41]:
```
# Get characters from 2 to 5 of "Coding"
c[1:5]
```

[41]: `'odin'`

[42]:
```
# Alternative way
c[-5:-1]
```

[42]: `'odin'`

[43]:
```
# Get characters from the first to the 4th
c[0:4]
```

[43]: `'Codi'`

[44]:
```
# The first index can be omitted!
c[:4]
```

```
[44]: 'Codi'
```

```
[45]: # Get characters from the 3rd to the last
      c[2:]
```

```
[45]: 'ding'
```

```
[46]: # Finally, give me the whole string!
      c[:]
```

```
[46]: 'Coding'
```

1.9 Manipulating strings: string methods

Some other times we want to manipulate textual data directly. Given a string we want to perform some operations that return the same string *slightly changed*.

This can be done using **string methods**. Loosely speaking, a **method** is a command that applies predefined transformations to an **object**, in this case a string.

The general syntax for **calling** a method is:

`string_name.method_name(<arguments>)`

There are *a lot* of string methods that we can use. Check here for a full list.

Below are a few examples:

```
[47]: # Capitalize a given string
      messy_string = "thIs IS A mESSy STrINg"
      messy_string.capitalize()
```

```
[47]: 'This is a messy string'
```

```
[48]: # Swap lowercase with uppecase
      messy_string.swapcase()
```

```
[48]: 'THiS is a MessY stRinG'
```

```
[49]: # Capitalise each separate word in a string
      messy_string.title()
```

```
[49]: 'This Is A Messy String'
```

```
[50]: # ALL UPPERCASE
      messy_string.upper()
```

```
[50]: 'THIS IS A MESSY STRING'
```

1.9.1 Searching methods

There are methods that allow you to search for specific text in a given string.

```
[51]: help(str.find)
```

```
Help on method_descriptor:

find(…)
    S.find(sub[, start[, end]]) -> int

    Return the lowest index in S where substring sub is found,
    such that sub is contained within S[start:end].  Optional
    arguments start and end are interpreted as in slice notation.

    Return -1 on failure.
```

```
[52]: # Find "fox" in the following string
      foxy = "the quick brown fox jumps over the lazy dog"
      foxy.find("fox")
```

```
[52]: 16
```

We can also replace existing text with new text.

This is done using the `str.replace()` method:

```
[53]: # Replace "dog" with "cow"
      foxy.replace("dog", "cow")
```

```
[53]: 'the quick brown fox jumps over the lazy cow'
```

1.9.2 Counting methods

In a lot of situations it is useful to know how long the string is or how many occurences of a certain text there are in a string.

This is done with two methods: - the `len()` method - the `str.count()` method

```
[54]: # Remember our long sentence? How long exactly was it?
      print(long_sentence)
      print(f"The sentence above contains {len(long_sentence)} characters!")
```

```
Sometimes you have a very long sentence that does not fit on a single line and
you might want to define it over several lines. Nevertheless this is stored, and
printed, as a SINGLE line
The sentence above contains 185 characters!
```

```
[55]: # How many occurences of "to" are there?
      long_sentence.count("to")
```

```
[55]: 2
```

2 Sequence types: lists, tuples, and dictionaries

We know how to store information in single variables. Numbers and text can be assigned to a variable using the two data types we introduced above: **Numbers** and **Strings**.

That's great but what if we have lots of them?

Sometimes it is useful to store information in a single place. This can result in a tidier code and less usage of many redundant variables.

Python can store multiples numbers and strings in so-called **sequence types**

- **Lists**
- **Tuples**
- **Dictionaries**

We will introduce and work with these data types in the next lecture.

3 Happy coding